AFRL-RZ-WP-TR-2011-2097

PRISMATIC: UNIFIED HIERARCHICAL PROBABILISTIC VERIFICATION TOOL

David J. Musliner and Eric Engstrom

Smart Information Flow Technologies, LLC

SEPTEMBER 2011
Final Report

AIR FORCE RESEARCH LABORATORY
PROPULSION DIRECTORATE
WRIGHT-PATTERSON AIR FORCE BASE, OH 45433-7251
AIR FORCE MATERIEL COMMAND
UNITED STATES AIR FORCE

NOTICE AND SIGNATURE PAGE

This report was cleared for public release by the USAF 88th Air Base Wing (88 ABW) Public Affairs (AFRL/PA) Office and is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RZ-WP-TR-2011-2097 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH THE ASSIGNED DISTRIBUTION STATEMENT.

*//Signature//
RYAN M. HISEROTE
Program Manager
Mechanical Energy Conversion Branch
Energy/Power/Thermal Division

//Signature//
JACK U. VONDRELL
Branch Chief
Mechanical Energy Conversion Branch
Energy/Power/Thermal Division

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
September 2011	Final	30 September 2010 – 30 September 2011

4. TITLE AND SUBTITLE
PRISMATIC: UNIFIED HIERARCHICAL PROBABILISTIC VERIFICATION TOOL

5a. CONTRACT NUMBER
FA8650-10-C-7077

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
62303E

6. AUTHOR(S)
David J. Musliner and Eric Engstrom

5d. PROJECT NUMBER
DM30

5e. TASK NUMBER
01

5f. WORK UNIT NUMBER
DM300100

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Smart Information Flow Technologies, LLC
211 North 1st Street, Suite 300
Minneapolis, MN 55401-2078

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Air Force Research Laboratory
Propulsion Directorate
Wright-Patterson Air Force Base, OH 45433-7251
Air Force Materiel Command
United States Air Force

Defense Advanced Research Projects Agency/Microsystems Technology Office (DARPA/MTO)
3701 Fairfax Drive
Arlington, VA 22203-1714

10. SPONSORING/MONITORING AGENCY ACRONYM(S)
AFRL/RZP

11. SPONSORING/MONITORING AGENCY REPORT NUMBER(S)
AFRL-RZ-WP-TR-2011-2097

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited.

13. SUPPLEMENTARY NOTES
Report contains color.

PA Case Number 88ABW-2011-5561; Clearance Date: 18 Oct 2011.

14. ABSTRACT
Research efforts were conducted under this task order to emphasize unique technologies in support of achieving the program goals associated with the DARPA META II Program. The contractor focused on technologies and technological breakthroughs addressing critical issues including scalability and abstraction in system verification, statistical verification, incremental verification, culprit identification, and counterexample generation. Collaboration with DARPA and other contractors on this program including PARC and Boeing was facilitated to optimize technology exchange and transition. Presentations were provided to government agencies, academia, and industry via workshops, seminars, and symposia.

15. SUBJECT TERMS
probabilistic verification, cyber-physical systems, statistical verification, incremental verification, culprit identification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT:	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON (Monitor)
a. REPORT	b. ABSTRACT	c. THIS PAGE			Ryan M. Hiserote
Unclassified	Unclassified	Unclassified	SAR	44	19b. TELEPHONE NUMBER *(Include Area Code)* (937) 255-2252

Table of Contents

Section **Page**

List of Figures

Figure **Page**

Acknowledgments

This work was supported by DARPA and Air Force Research Laboratory under contract FA8650-10-C-7077. Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of AFRL or the U.S. Government.

1.0 PROJECT SUMMARY

Smart Information Flow Technologies (SIFT) LLC, Carnegie-Mellon University, and Oxford University have developed PRISMATIC, a unified tool and technique for formal design verification to address the challenges of verifying complex cyber-physical system designs before manufacturing and testing. The extreme complexity of modern cyber-physical systems such as ground combat vehicles and military aircraft, makes their design and manufacture very difficult and time consuming. The META program aims to improve the process of building cyber-physical systems by developing new model-based design flows and tools that can capture all functional and logical aspects of a system design, allowing design-time verification of system behavioral properties.

Verification of entire cyber-physical systems poses several key challenges, including heterogeneous designs that span cyber and physical domains; uncertainty in component reliability and partially-sensed, uncontrollable environments; and large-scale designs with operating state spaces far larger than can be handled by existing verification approaches. To meet these challenges we developed PRISMATIC, a workflow/process and tool that can generate probabilistic "certificates of correctness" for entire large-scale cyber-physical systems such as ground combat vehicles or aircraft. Our team had previously developed several ground-breaking verification methods and state-of-the art open-source verification tools. We built on these existing tools and develop unique new capabilities for scaling through compositionality to meet the challenges of cyber-physical system design and manufacturing flow.

The core of PRISMATIC is based on Oxford's PRISM system, already the leading probabilistic verification tool in terms of functionality, efficiency and adoption. PRISM's analytic probabilistic verification techniques work well for some types of models, but can be overwhelmed by very large state spaces. SIFT and Carnegie Mellon had developed statistical probabilistic verification techiques that do not rely on analytic state space reasoning, but reason adaptively about Monte Carlo samples (traces) of system execution. For PRISMATIC we scaled these statistical verification techniques to much larger state spaces, and have the added advantage of needing only a system simulation or testbed (rather than a formalized, analyzable model). Thus PRISMATIC can verify hybrid cyber-physical systems through integration with other simulation frameworks, such as the engineering industry standard MATLAB® Simulink® system. Carnegie Mellon has also provided non-probabilistic, BDD-based and SAT-based verification capabilities that can be used to verify properties of the digital components of cyber-physical systems. In addition to these complementary exact and probabilistic verification techniques within the tool itself, PRISMATIC also reasons about other design domains such as mechanical and thermal models by interfacing to existing domain-specific verifiers.

We developed several novel techniques to address specific verification challenges:

- **Compositional, hierarchical** reasoning that decomposes an overall system verification problem into a combination of subproblems, mitigating the state space explosion problem.
- **Counterexample and culprit identification** that provides debugging and re-design guidance to users when a system design is not verified to meet its requirements.
- **Incremental verification** that preserves partial verification results, dramatically speeding up re-verification after design changes in the design and redesign cycle.

The resulting PRISMATIC tool provides a verification flow for cyber-physical systems that reasons about entire cross-domain designs and is optimized for its role in the iterative design-verify-revise cycle. PRISMATIC should reduce system verification time by at least a factor of five, enabling the overall META program to dramatically reduce the cost and time for cyber-physical system development.

Our goals for this phase were that PRISMATIC should efficiently perform several key probabilistic verification functions on a given complex cyber-physical system design:

- Calculation of a probabilistic "certificate of correctness" that the system satisfies the design requirements to a given level of confidence.
- Conversely, calculation of the required design component behavioral constraints that will achieve a desired system-level certificate of correctness.
- Calculation of the probability that design rules or manufacturability constraints are met.
- Rapid re-verification of a system after configuration changes (e.g., due to component failures or adaptation), via incremental, modular verification processes.
- Estimation of the cyber-physical system's reliability, with minimal real-world testing.

PRISMATIC will have enormous impact the overall cyber-physical system design process, by making formal verification a practical tool to use frequently, as part of the daily iterative design cycle. PRISMATIC can verify the link between system designs and their formalized requirements, and actively guide designers in modifying their systems when requirements are not met. Furthermore, PRISMATIC dramatically reduces the need to physically build and test system components to ensure proper operations. And when real-world testing of implemented prototypes is needed, PRISMATIC's statistical verification methods helps minimize the testing required.

2.0 INTRODUCTION

This report describes the results of the Phase 1 PRISMATIC effort for the META program. The META program is improving the design of military cyber-physical systems by developing new model-based design flows and tools that can capture functional and logical aspects of a system, allowing design-time verification of system behavioral properties. Many cyber-physical systems are used in military applications where mission failure or loss of life is a significant possibility. For example, in 1992 an F-22 jet crashed because of an error in the flight control software [34]; in September 1997 the USS Yorktown was left at sea with no power for several hours because bad sensor data was entered into an onboard computer system and handled improperly [38]. There faults are clearly unacceptable. Our ability to assemble and deploy large-scale cyber-physical systems has exceeded our ability to ensure their correctness. To avoid such failures, it is of paramount importance to determine that cyber-physical systems actually meet the goals for which they are designed.

To address these challenges we are developing PRISMATIC, a workflow/process and tool for generating probabilistic "certificates of correctness" for entire large-scale cyber-physical systems such as ground combat vehicles or aircraft. In designing and implementing PRISMATIC our team has extended several of our previous ground-breaking verification methods and state-of-the art open-source verification tools, and developed unique new capabilities for scaling through compositionality to meet the challenges of cyber-physical system design and manufacturing flow.

Automatic design verification techniques are intended to check that a particular system design meets a set of formal requirements. Verification offers the promise of dramatic reductions in the cost and schedule of complex system development projects, by improving several aspects of the design and testing process:

- More flexible requirements allocation.
- Early recognition of design flaws and interactions.
- Guided design revisions.
- Rapid re-verification after system design changes or adaptation.
- Pre-testing assessment of system reliability, with error bounds, accounting for single- and multi-mode failures.
- Reduced need for testing.

Verification of complete cyber-physical systems poses several key DARPA-hard challenges, including:

- **Heterogeneous, cyber-physical domains** — Tight integration of physical components such as vehicle control mechanisms with computing hardware and software produces unique verification problems that span a variety of design domains. Many domain-specific design and verification tools are available, but they typically do not allow cross-domain analysis. For verification of physical components, a typical problem is to ensure that a component can sustain 150% load. For military flight software, verification is concerned with the correctness of software according to the requirements. To produce a system-level certificate of correctness or safety, we must develop new formal methods for combining probabilistic certificates of correctness of heterogeneous components and their interactions.

- **Uncertainty** — Traditional complete verification methods work well for confirming device functional behaviors, when they apply (e.g., to structured VLSI components), but many cyber-physical systems necessarily include fallible system components (e.g., failure-prone mechanical devices) and interactions with an uncertain, imperfectly-sensed, and only-partially-controllable environment. These aspects necessarily introduce uncertainty in the verification process, requiring new verification techniques. Our team includes leading researchers in probabilistic verification, who have developed the current state-of-the-art verification algorithms and tools.
- **Large scale** — Currently, the best available non-probabilistic model checking verification systems are easily overwhelmed by the complexity of even fairly small software systems. Probabilistic model checking systems have been demonstrated on systems with as many as 10^{17} possible states [21], but even this power is dwarfed by the potential state space size of a full cyber-physical system. New techniques must be developed to divide-and-conquer the overall system verification problem, so that smaller components and subsystems can be verified to a given level of reliability, and those results combined into higher-level, system-wide probabilistic certificates of correctness. In addition, efficiency techniques such as incremental verification and parallelization may be leveraged to improve the scalability of verification methods.

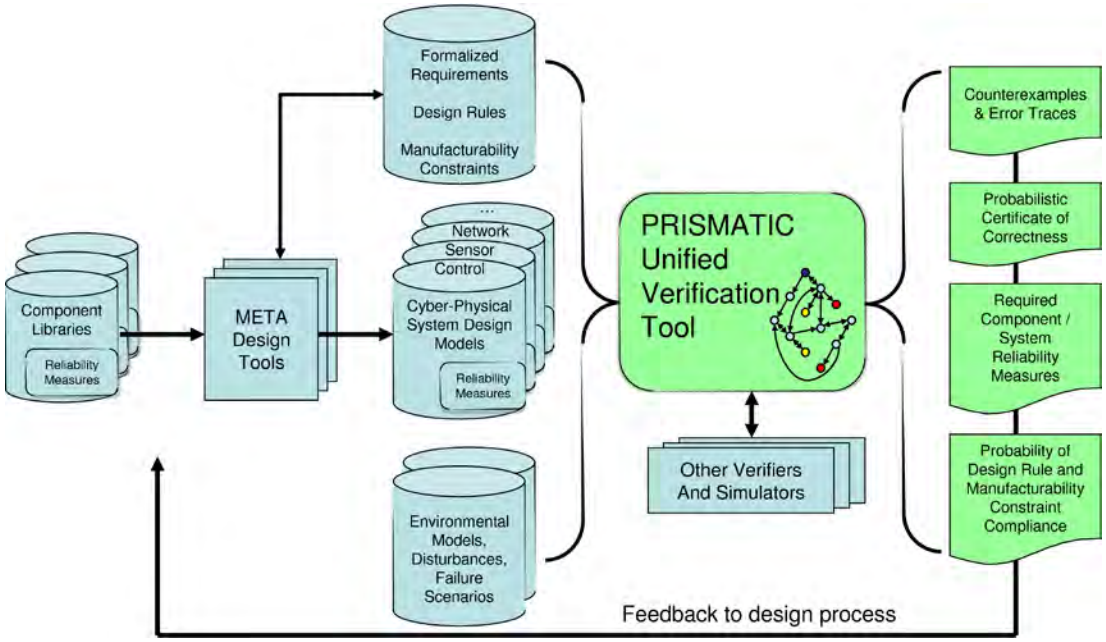

Figure 1: Cyber-physical System Design with PRISMATIC

The PRISMATIC tool provides rapid, incremental verification of full cyber-physical system designs to produce probabilistic certificates of correctness or counterexamples and related feedback that can be used to revise design elements and component libraries.

Figure 1 illustrates how our PRISMATIC operates within an iterative, incremental design-verify cycle for cyber-physical systems. PRISMATIC interfaces to design tools that produce multi-domain models capturing various aspects of a system design, such as its mechanical elements, computing hardware, and software. PRISMATIC consumes formalized specifications of

the system requirements that it is supposed to verify (e.g., safety constraints); these may be captured within the design tools or in separate requirements-capture tools. Finally, PRISMATIC takes in models of the cyber-physical system's expected mission and operating environment, including disturbance models and component failure scenarios. From these inputs PRISMATIC verifies individual components and subsystems, combining them using new compositional algorithms to verify the entire system. Or, when the verifier proves that a system design does not satisfy its requirements, PRISMATIC returns counterexamples to the design tools, helping guide debugging and design revisions.

Our team had already developed several ground-breaking verification methods and implemented state-of-the art verification tools that were leveraged to build PRISMATIC. In particular, the Oxford team has been developing the PRISM probabilistic model checking system for over a decade, and its extensible architecture forms the core of PRISMATIC. We have added new types of built-in verification, and we have extended its interfaces so that it can coordinate the operations of other domain-specific verifiers and simulators.

As a result of our work PRISMATIC can offer several important benefits, including:

- **Scalability** — PRISMATIC scales to verify extremely complex systems via compositional verification, incremental verification, and statistical methods that scale well with model size.
- **Speed** — PRISMATIC verifies new system designs quickly using a compositional, divide-and-conquer strategy. PRISMATIC takes advantage of parallel processing, since statistical methods are parallelizable and support concurrent verification of numerous system properties.
- **Rapid re-verification** — Changing a cyber-physical system design may not require complete re-verification, because PRISMATIC's compositional and incremental verification techniques minimize the need to re-verify unchanged components.
- **Design guidance** — PRISMATIC helps guide debugging and system re-design efforts by identifying culprits and deriving requirements on future design revisions that will move a system closer to compliance with desired safety or behavioral specifications.
- **Generality, broad applicability** — PRISMATIC's statistical verification methods work for any type of system that can be simulated, using any form of probability distributions. Thus PRISMATIC can easily combine results from different cyber-physical design disciplines.

PRISMATIC can have enormous impact on the overall cyber-physical system design process, by making formal verification a practical tool to use frequently, as part of the daily iterative design cycle. PRISMATIC can verify the link between system designs and their formalized requirements, and can actively guide designers in modifying their systems when requirements are not met. Furthermore, PRISMATIC can dramatically reduce the need to physically build and test system components to ensure proper operations. When real-world testing of implemented prototypes is needed, PRISMATIC's statistical verification methods can help minimize the testing required.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

The primary objective of the Phase 1 effort has been to develop a PRISMATIC tool that efficiently performs the key probabilistic verification functions on complex cyber-physical system designs described in the introduction. Achieving this objective first required SIFT, Carnegie Mellon University (CMU), and Oxford to achieve a number of research goals:

- Unifying probabilistic and non-probabilistic verification.
- Investigating hybrid, cross-domain verification methods.
- Applying compositional reasoning in verification.
- Developing incremental verification techniques.
- Enhancing analytic verification with counterexample generation.
- Effectively simulating rare events via importance sampling.

We completed these implementation and research goals as proposed through the following specific tasks, which we describe in detail in the sections below.

- Creating clearly written requirements, use cases, and an evolving design for the PRISMATIC tool (Section 3.1).
- Implementing statistical verification algorithms for the unified PRISMATIC tool (Section 3.2).
- Implementing verification methods to define and exploit system decomposition, allowing PRISMATIC to combine probabilistic certificates of correctness from components or subsystems into higher-level probabilistic verification results (Section 3.3).
- Researching and implementing methods for incremental verification, allowing rapid analysis of models after changes (Section 3.4).
- Implementing an approach for identifying culprits, the individual internal components responsible for failures (Section 3.5).

3.1 PRISMATIC Design

Figure 2 shows a breakdown of the PRISMATIC architecture into functional components. The PRISMATIC architecture leverages the components and architecture of Oxford's PRISM probabilistic model checking system: PRISM's extensible architecture forms the core of PRISMATIC. In the following paragraphs, we will briefly review the structure of PRISM, and then explain how PRISMATIC extends the core system with new capabilities.

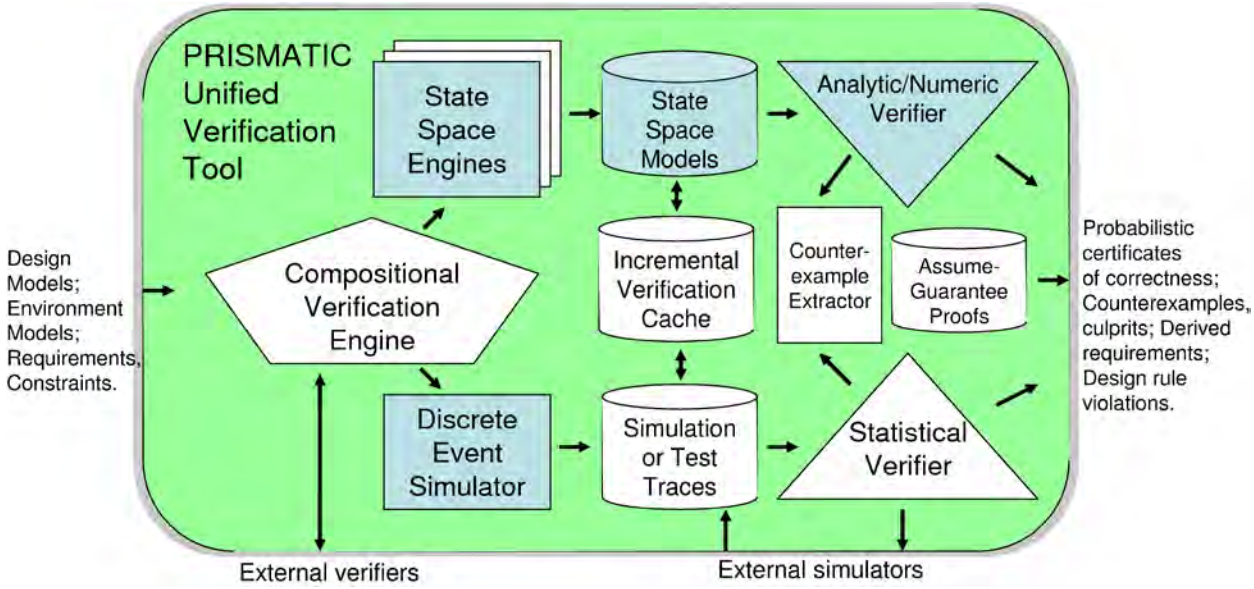

Figure 2: PRISMATIC Architecture

PRISMATIC's compositional reasoning divides system verification problems into smaller verification tasks that can be addressed by analytic or statistical verification methods. PRISMATIC leverages and extends the existing PRISM tool (shaded boxes), also adding incremental verification methods for efficient iterative design and verification.

3.1.1 PRISM Overview

Oxford has been developing the open source PRISM probabilistic model checking system for over a decade, and it is now the leading tool for analytic/numeric probabilistic verification. PRISM provides support for modeling and analysis of several different classes of probabilistic models: discrete- and continuous-time Markov chains, Markov decision processes and probabilistic timed automata. It provides a flexible language for describing such models and supports a wide range of probabilistic temporal logics for specifying properties to be verified. The tool has been applied to more than fifty real-world case studies covering a broad spectrum of application areas: randomized distributed coordination protocols, including IEEE 1394 Firewire; wireless protocols, such as Bluetooth device discovery; security protocols such as for anonymity and quantum cryptography; and biological reaction pathways.

PRISM is currently the leading probabilistic verification tool in terms of functionality, efficiency and adoption. It is used for teaching and research in more than fifty institutions worldwide and has been downloaded more than 23,000 times. There are over 165 external and 125 internal PRISM publications and research projects centered around the tool have received funding from both UK and EU research councils (EPSRC, ERC) and industry (including Microsoft Research, QinetiQ, and BT).

PRISM incorporates multiple efficient engines, primarily based on symbolic model checking techniques that use extensions of binary decision diagram (BDD) data structures. Current PRISM components are shown shaded in Figure 2: *State Space Engines* translate design models into efficient internal data structures, the *State Space Models*. Using these state space models, the *Analytic/Numeric Verifier* checks the truth of claims about the system, for example integrating

probability information over classes of safe trajectories to compute lower bounds on the probability of safe system function. In addition, PRISM includes a *Discrete-Event Simulator*, which can be used to generate sample executions through PRISM models, providing a basis for the integration of SIFT and CMU's statistical model checking techniques (Section 3.1.2, below).

PRISM's modular, open architecture is a good foundation for PRISMATIC: it already has a proven track record of being successfully extended to implement novel verification techniques by both the PRISM team and by external research groups. For example: recent work on compositional probabilistic verification by Prof. Kwiatkowska's team [11, 22] was developed as an extension of PRISM; researchers at the University of Konstanz have developed heuristics-based search techniques [3] on top of PRISM's discrete event simulation engine; and researchers at Saarland University, Saarbrücken extended PRISM with techniques to analyze infinite-state Markov chains [18]. The scope and applicability of PRISM have also been extended through various connections to external tools and design languages. For example, through language-level translations to its modeling language, PRISM offers indirect support for SBML (Systems Biology Markup Language) and probabilistic extensions of various process algebras (π-calculus, PEPA, CSP). Conversely, the PRISM language is used as input to several external tools, such as the statistical probabilistic model checkers YMER and APMC, and research prototypes for abstraction (PASS, INFAMY) and parametric verification (PARAM). PRISM also interfaces with other model checkers, such as ETMCC and MRMC, and numerical software such as MATLAB.

3.1.2 From PRISM to PRISMATIC

The PRISMATIC components that we developed in META are shown unshaded in Figure 2. The front end for PRISMATIC is the *Compositional Verification Engine,* which consumes incoming design models, and manages the verification flow. This component manages the *assume-guarantee* reasoning needed to divide-and-conquer large-scale verification problems. The assume-guarantee approach requires deriving guarantees about some subsystem's function and using those guarantees as "assumptions" later to derive further guarantees of sibling or parent components. Thus a key question is what assumptions and subsystem partitions should be our focus. The compositional verification engine captures decomposition information inherent in the cyber-physical system designs, and also incorporates novel adaptive decomposition techniques to automatically find the decompositions that are most productive of design and verification savings. Designers may also use the assume-guarantee functions "in reverse" to derive the performance bounds on a component that are required to guarantee a performance bound on the overall system. That is, showing that *if* the overall system must exhibit some desired property *P, then* a particular subsystem must provide a property *P′* with at least some probability.

PRISMATIC also integrates analytic/numeric and newer statistical methods for verifying probabilistic systems, adding the *Statistical Verifier* component shown in Figure 2. To date, PRISM uses symbolic and numeric reasoning over system specifications to deduce whether or not properties will be satisfied. In contrast, work by CMU and SIFT has focused on verifying properties by statistical model checking based on simulation of a system design. We contrast these two techniques below in Section 4.2.

We have also added interfaces from PRISMATIC to alternative, domain-specific verifiers so that their results can be incorporated into the overall verification results. The compositional verification engine merges assume-guarantee results into unified proofs stored in the Assume-Guarantee Proof database.

In a sense, the *counterexample extractor* component performs a task opposite to that of assume-guarantee reasoning. Assume-guarantee reasoning allows us to use the verification of individual subsystem properties as building blocks to verify overall system properties. The counterexample extractor guides designers' progress when PRISMATIC *fails* to verify a property (or fails to verify it to a desired degree of probability). In conventional (non-probabilistic) verification systems, counterexample extraction is simply a matter of finding a single trace that violates a claim. However, in probabilistic systems it is not sufficient, in general, to find a single counterexample to violate a property. Instead, we must find a class of counterexamples with some amount of probability mass sufficient to violate the likelihood threshold in the claim. We discuss how to find and exploit such classes of counterexamples in Section 4.5.

To support the suite of components in the verification flow, PRISMATIC includes a database of *Simulation and test traces* and an *incremental verification cache.* The trace database is harvested by the counterexample extractor after a failed verification, to guide PRISMATIC users in redesign. SIFT's incremental verification techniques use the verification cache (see Section 4.4) to speed subsequent verifications, as does compositional reasoning.

3.2 Statistical Verification

Previous verification techniques can be divided into two basic classes: probabilistic and non-probabilistic approaches. Non-probabilistic verification works on system models that have no uncertainty, and is used to formally prove operating properties based on those models. Team member Prof. Edmund Clarke of CMU pioneered formal verification through *model checking,* a technique that has led to revolutionary advances in VLSI circuit design complexity, allowing system designs with millions of transistors to be automatically checked for proper performance. Prof. Clarke shared the 2007 Turing Award for developing this technology. However, when applied to the more complex domain of software verification, pure model checking methods have stumbled—they do not scale as well on the less-structured designs that arise in software systems.

Furthermore, in many real-world system designs, there is some non-zero probability of failure or requirements violation. Such stochastic behavior arises naturally, for example because of uncertainties present in a system's components or its environment (e.g., the reliability of communication links in a wireless sensor network, the rate of message arrivals on an aircraft's communication bus). In these domains, the goal of verification is to check that the actual probability of a failure occurring, while non-zero, is acceptably small. For example, we want to know whether the probability of a communication bus delaying a message is smaller than 0.00001; or whether the system fulfills a request within one millisecond with probability at least 0.9999.

The purely logical proofs used in traditional model-checking cannot apply to such systems; instead, *probabilistic model checking* (PMC) methods are required. These require the construction of a stochastic model of the system to be verified and the specification of its requirements in probabilistic temporal logic [4, 5, 16]. The problem is then to decide whether the model satisfies a temporal logic property with a probability greater than or equal to a certain threshold. More formally, suppose M is a stochastic model over a set of states S, s_0 is a starting state, φ is a formula in temporal logic, and $\theta \in (0,1)$ is a probability threshold. The PMC problem is to decide algorithmically whether model M satisfies property φ with probability at least θ; in symbols, $M, s_0 \models P_{\geq\theta}(\varphi)$.

Early work on probabilistic model checking focused on extending traditional analytic model checking techniques with numeric methods to determine the probability that a property is satisfied. In addition to exhaustive exploration and construction of a stochastic model, probabilistic model checking requires the numerical solution of, for example, linear equation systems or linear optimization problems. Considerable progress has been made in developing efficient implementations of these techniques, resulting in powerful probabilistic model checking tools, such as PRISM [19]. However, numerical solution remains computationally demanding, and as with all analytic model-checking methods, the approach scales poorly as the system's reachable state space grows large (e.g., above $10^7 – 10^8$ states). In many real-world systems, the number of states can easily exceed this limit. This necessitates the development of alternative, more scalable methods.

3.2.1 Statistical Probabilistic Verification

One promising approach to probabilistic model checking, which Dr. David J. Musliner (SIFT) and Prof. Clarke have been investigating, is *statistical model checking* [14, 40, 41, 36]. This technique operates quite differently from analytic/numeric approaches, providing many of the same advantages with far better scalability. Rather than numerically propagating uncertainty through a detailed system model, statistical model checking algorithms simply draw sample traces of the system's execution (e.g., from a simulation or test rig) and use statistical methods to test whether the accumulated samples justify a sufficiently-confident conclusion about whether the system's performance meets the requirements. If a conclusion is not yet possible, more samples are drawn. This approach is less sensitive to the state explosion problem that causes difficulties for exhaustive model checking algorithms.

These techniques rely heavily on simulation, which is generally easier and faster than a full symbolic study of the system. This may be an important factor for industrial and defense cyber-physical systems, which are often designed using efficient simulation tools like MATLAB SIMULINK™ [1]. Since all we need are simulations of the system, we do not have to translate system models into a separate modeling language for a verifier, nor do we have to build symbolic models of the system (e.g., Markov chains) suitable for numerical methods. Eliminating these intermediate representations not only speeds up the verification process, it eliminates another source of potential error: there is no separate verification model that can become inconsistent with the system model.

Statistical model checking treats the probabilistic model checking problem as a statistical inference problem, and solves it by randomized sampling of the *traces* (or simulations) from the model. The traces are model checked to determine whether the desired property holds, and the number of satisfying traces is used to decide (approximately) whether $M, s_0 \models P_{\geq\theta}(\varphi)$. The decision is made by means of either *estimation* or *hypothesis testing*. In the first case one seeks to *estimate probabilistically* (e.g., compute with high probability a value close to) the probability that the property holds and then compare that estimate to θ [17, 36] (in statistics, such estimates are known as *confidence intervals*). In the second case, the probabilistic verification problem is directly treated as a *hypothesis testing* problem, that is, deciding between two alternate hypotheses—$P_{\geq\theta}(\varphi)$ versus $P_{<\theta}(\varphi)$ [40, 41, 42]. Hypothesis-testing-based methods are more efficient than those based on estimation when the probability threshold θ (which is specified by the user) is significantly different from the true probability that the property holds (which is determined by M, s_0, and φ)

[39]. On the other hand, it might be necessary to give an accurate approximation of the unknown probability that the property holds, and that is just what estimation methods aim to do.

Our statistical model checking approach for both hypothesis testing and estimation is based on Bayes' theorem and sequential sampling. Bayes' theorem enables us to incorporate prior information about the model being verified, where available. Sequential sampling means that the number of sampled traces is not fixed *a priori,* but it is instead determined dynamically at "run-time," depending on the evidence gathered by the samples seen so far. This often leads to significantly smaller number of sampled traces (simulations). The number of samples in the sequential sampling scheme can be three orders of magnitude smaller than fixed sampling schemes. Our proposed estimation method follows directly from our Bayesian approach. In fact, Bayes' theorem enables us to obtain the *posterior* distribution of the true probability p with which the model satisfies the formula (i.e., the distribution of p according to the data sampled and chosen prior). By integrating the posterior over a suitably chosen interval, we can compute a Bayes interval estimate with any given confidence coefficient.

Statistical verification methods have proven to scale very well as the size of the system model and reachable state space grows. For example, Figure 3 (from [39]) shows two graphs that compare the performance of analytic/numerical and statistical model-checking methods. As shown, the numerical methods scale poorly, with verification time growing exponentially as the state space grows. In contrast, the statistical approach can scale almost independently of system size; roughly speaking, statistical methods take longer as the difficulty of the question being answered grows. Specifically, when the *actual* probability that the system design satisfies or violates the desired property φ is close to the verification threshold θ, statistical methods have more trouble reaching a conclusion, and require more samples.

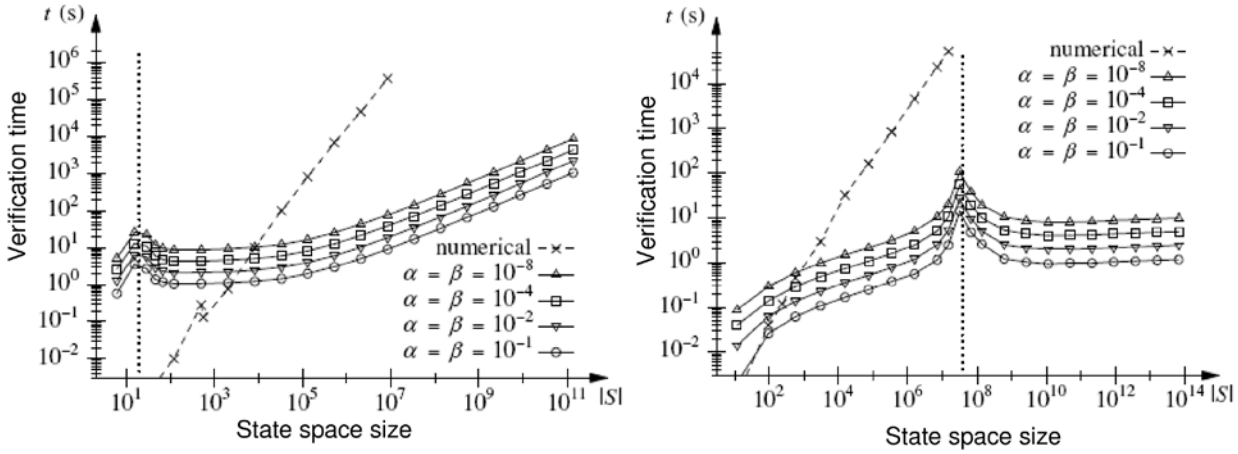

Figure 3: Verification Time Comparison for Analytic/Numerical and Statistical Model Checking Methods

In addition, because statistical methods do not need to represent the entire system state space, nor must they retain the execution traces they assess, their memory usage can be very low (near-constant). In contrast, numerical methods need exponentially increasing memory, and often fail by running out of memory before they exceed the allotted verification time.

We have developed statistical model checking techniques and applied our Bayesian hypothesis testing approach to very large (10^{17} states) stochastic systems, where it has outperformed other

statistical techniques by an order of magnitude [21]. We have also had initial success with both our Bayesian hypothesis testing and estimation methods on examples of discrete-time, stochastic cyber-physical system coded as Simulink/Stateflow diagrams, and we are observing similar speed-ups [44].

For even greater speed and scalability improvements, statistical methods may be deployed on parallel processors. Statistical model checking methods are easier to parallelize than exhaustive verification techniques, because each sample trace can be generated independently. In fact, Younes' open-source YMER [39] tool already includes support for distributed acceptance sampling; tests show a performance improvement close to linear in the number of CPUs made available to YMER. We plan to investigate parallelization and distribution of Statistical Model Checking directly for the problem domain of Simulink/Stateflow models.

Statistical methods can make use of additional memory to store partial or complete traces and use them to provide additional benefits, including design revision guidance/advice and more rapid verification of revised designs. For example, a statistical verification system may preserve those traces that lead to failures of various kinds and analyze them to let a designer know which system components are most responsible for an unacceptable failure rate. Or, all of the traces may be preserved (subject to some arbitrary memory limits) and the verification engine can re-use those traces or trace prefixes that are not affected by a design component that has been changed.

Statistical model checking is not limited to pure offline simulation. It is applicable to hardware-in-the-loop simulations and live testbeds, and might even help by increasing confidence in system designs during the initial system operation phase by combining runtime verification with Statistical Model Checking techniques.

However, one problem with statistical model checking is caused by rare events, *i.e.* properties whose probability of being true is small enough relative to the number of samples that can be generated or evaluated that we are unlikely to be able to observe enough examples (or even one example) of the rare event to make a reliable estimate of its probability. It is well known that straight Monte Carlo techniques such as statistical model checking do not perform well when estimating rare-event probabilities. The problem is that the sample size (*i.e.* the number of simulations) required for accurate estimates grows too large as the event probability tends to zero. However, several techniques have been developed to address this problem.

We have been studying importance sampling techniques, which bias the original system to increase the likelihood of the event of interest, and then weight the simulation results in order to obtain unbiased estimates. The main difficulty in importance sampling is to devise a good biasing density, that is, a density yielding a low-variance estimator. Optimal, zero-variance biasing densities do exist, but are extremely difficult to sample from.

We have been investigating two techniques for variance reduction with importance sampling. Both techniques search for a biasing density in a parameterized family of densities which include the original density of the system. The first technique is the cross-entropy method, which searches in the parameterized family of densities for a biasing density "close" to the optimal one. A practical notion of "closeness" between two densities is provided by the cross-entropy (or Kullback-Leibler divergence). The second technique seeks to (numerically) find the density which actually minimizes the variance of the importance sampling estimator.

We have used the cross-entropy method [8] and variance minimization [43] for generating optimal biasing densities in statistical model checking. In particular, we have applied both techniques with importance sampling for verifying Stateflow/Simulink models of a fault-tolerant fuel control system and of a fault-tolerant controller for an aircraft elevator system. Our results suggest that importance sampling and can be successfully combined for statistical model checking of rare events in moderately large, though realistic, cyber-physical systems.

3.3 Compositional Verification

Although statistical verification methods can scale to very large state spaces, there are still significant challenges in applying them to real-world cyber physical systems. Capturing the interactions of hundreds of components is beyond the state of the art for a single verification operation. However, Oxford's recent research in probabilistic "assume-guarantee" verification shows promise for truly compositional verification—that is, a means by which separate verification results for components can be combined to give a system-level verification result [22]. Using assume-guarantee methods, we verify a set of requirements on one component and then show that, if those results hold (the assumption), then a second component operating in the environment of the first will meet a desired overall performance guarantee. For example, if we assume that component X fails with a probability no higher than p, then we can guarantee that the system as a whole will only deadlock and fail with a probability $\leq p/2$. The assumption allows us to partition the overall system verification problem into two smaller problems, reducing the computational burden.

However, deciding what assumptions should be used to partition the system is currently an art. In the year prior to META, Oxford had made progress on using algorithmic learning techniques, already shown to be successful in non-probabilistic compositional verification [6, 26], to automatically generate such assumptions [11]. Our research has extended and implemented this approach in PRISMATIC. Previously, assume-guarantee reasoning using PRISM was a manual operation in which the user of the tool is required to (1) select the appropriate abstraction points, (2) break the problem into multiple models, and (3) perform the separate proof steps of assume-guarantee reasoning in a non-automatic manner. The Oxford Group has now completed the additional input language syntax and mechanisms within PRISMATIC to automatically perform assume-guarantee reasoning (Step 3 above). Automatically deriving assume-guarantee partitions would thus combine results from both PRISMATIC's own analytic and statistical verification engines as well as external verifiers (e.g., NESSUS [2]).

Another key challenge in making this technique practical was using the system design model to understand which behaviors and failure modes are independent across decomposition, and which are dependent or related. For example, two subsystems that are attached to the same power bus may have independent failure characteristics related to their internal components or software, and dependent failure rates related to transients on the shared power bus. Probabilistic certificates for these subsystems must carry along the assumption information that describes the environmental assumptions in which the guarantees hold true, so that the overall composite system can be reasoned about correctly.

3.3.1 Assume-Guarantee Reasoning with Statistical Verification

The assume-guarantee reasoning approach, as currently implemented in PRISM, is limited to analytic propagation of probabilities. Models which are too large for the analytic approach may be

amenable to statistical sampling approaches for inferring probabilities. However, there is a theoretical disconnect between the assume-guarantee reasoning and probability inferencing via statistical sampling. For example, consider the asymmetric proof rule:

$$\frac{\langle T\rangle M_1\langle A\rangle_{\geq pA} \quad \langle A\rangle_{\geq pA} M_2\langle AG\rangle_{\geq pG}}{\langle T\rangle M_1 \| M_2\langle G\rangle_{\geq pG}} \tag{1}$$

Existing sampling approaches such as simple packet relay transport (SPRT) can be used to determine whether the assumption, $\langle T\rangle\ M_1\langle A\rangle_{\geq pA}$, is satisfied. One challenge lies in generating samples that agree with the assumed property while providing a sufficient distribution to prove the guaranteed property. A statistical solution would sample some instances where $\langle A\rangle$ is satisfied, and some where it is not. Sample selection could be used to achieve the right balance of $\langle A\rangle$ vs. $\langle \bar{A}\rangle$ samples. A further issue is nondeterminism, which arises in the assume-guarantee reasoning formulation due to the composition operator. A nondeterministic scheduler is assumed to resolve races between concurrent action choices. Existing non-statistical solution methods use linear programming to optimize an adversary that acts in worst-case manner to defeat the property being verified. Finally, the assume-guarantee rule is incomplete, in the sense that although the system might satisfy the property, one will still not be able to not come up with an appropriate assumption (written as a probabilistic safety property) that makes both premises satisfiable.

The CMU team has experimented with *(strong) probabilistic simulation* [35] and proposed the following assume-guarantee rule for checking probabilistic simulation. Here M_1, M_2 and A are (labeled) probabilistic automata, and || denotes the parallel composition operator:

$$\frac{\begin{array}{ll} 1: & A||M_1 \sqsubseteq S \\ 2: & M_2 \sqsubseteq A \end{array}}{M_1||M_2 \sqsubseteq S} \tag{2}$$

The rule is both sound and complete; completeness follows simply from the fact that one can replace A with M_2 in the rule premises. Note however that for the rule to be useful in practice, assumption A needs to be much smaller than than M_2 but still reflect M_2's behaviour in such a way that the Premise 2 is satisfied. Coming up with such assumptions manually is a highly non-trivial exercise. Following seminal work by Pasareanu *et al.* [12, 26] that was done in the context of non-probabilistic verification, we have defined two new algorithms for building assumptions automatically for the compositional probabilistic verification using the rule above. In particular, this approach incorporates:

- *A new definition of (sub) stochastic tree counterexamples for the simulation conformance checks.* Previous work on simulation checking did not define counterexamples, although they are essential for model checking and debugging systems. We use counterexamples to automatically infer and refine assumptions used in compositional verification.
- *Assume-Guarantee Abstraction Refinement (AGAR) for simulation conformance.* The initial assumption is built as a conservative abstraction of M_2 (hence Premise 2 holds by construction). This abstraction is then iteratively refined based on counterexamples obtained by model checking Premise 1 iteratively. The process is repeated until either the conformance relation is shown to hold or a discrepancy between the system and the specification is found.

- *A new approach for learning the* minimal separating automaton *for languages of stochastic trees.* We have developed two variants: (i) learning minimum consistent partition and (ii) learning minimum state separating automata.

We have implemented the algorithms for simulation checking and generation of tree counterexamples, and subsequently refined the former based on the Yices SMT solver. We have also implemented the AGAR compositional algorithm, and began implementing the learning algorithms. We have applied these algorithms to the verification of the ramp subsystem. In order to apply our techniques, we had to change the communication from formula-based to proper synchronization, based on common actions. Such transformation would be needed for any compositional or hierarchical reasoning. We did a systematic, albeit manual, translation. We identified inputs coming from the other components and corresponding outputs. We created a communication action for each condition on these inputs (module $ElectricalActuator_{EA1}$) and we added transitions for the corresponding outputs according to the formulas. This implementation is able to verify system-level properties of the Ramp system in reduced time, and in greatly reduced space, than in previous simulation checking.

3.3.2 Abstraction Verification

Compositional verification depends on the assumption that system-level verification produces the same results when using the abstract components representing assume-guarantee partitions as it would if the verification could be performed using an expanded model incorporating all of the component-level detail. However, in order to be able to be able to realize the benefits of compositional verification (verification of larger models, reduced duplication of verification effort), there must exist a way to verify that a given abstraction accurately represents the relevant properties of the underlying component without needing to perform the entire system-level verification using both representations of the component. This requirement is especially necessary if we wish to enable system modelers to use third-party libraries of system component models without needing to perform expensive system-level verification to verify each component abstraction.

PRISMATIC uses component mode information (nominal and failure modes) defined in the detailed component model to perform this verification. When verifying that a given model is a valid abstraction of a specified detailed model, PRISMATIC checks the specified models for inputs or outputs that are unconnected (to other subcomponents of the component) and uses the values of those flows to analytically verify that the behavior of the abstract and detailed models match exactly in each component mode. In order to pass verification, an abstraction must always produce the same output as the detailed component model for the same input values in all modes, and it must enter each off-nominal mode with *at least* the same probability as the detailed model. An abstraction that enters off-nominal states with a greater probability than a detailed model is accepted, since properties depending on the correct functioning of the component that are verified against a system using the abstract model will also hold for the potentially more reliable detailed model.

The PRISMATIC *abstraction verification tool* (ABV) creates a series of tests by setting the abstraction next to the detailed input model to evaluate that they, indeed, have the same behavior. The ABV tool will create default formulae for comparing the modes of the abstract and detailed models. Should these defaults not capture the appropriate composition of detailed modes ABV also supports explicitly setting “mode combinations” using either conjunctive or, in the case of a

redundant system, disjunctive mode operators. In especially complex systems the actual expression to verify that both the abstract and detailed modes are in equivalent modes can be given explicitly as an expression.

In designing a realistically complex system one can leverage abstraction verification to perform modular decomposition of the system design. The PRISMATIC toolset includes the parallel abstract verifier (PABV) which has been designed to handle immense workloads. Given a list of available PRISMATIC web service instances PABV manages to keep the job queue of each web service full with at most two tasks in order to avoid overwhelming network and operating system resource limits. PABV will submit a ABV task for each compositional verification job submitted in parallel, gather the result values (including performance metrics) and return the rolled up system verification result. In this way proper assessments of the probabilistic behavior of the entire system can be performed by composing verified subsystems where each subsystem is an abstraction for the detailed components it comprises. This is essential as probabalistic model checking for a full system design is infeasible due to the combinatorial amount of computer memory required to represent the state space with enough multi-terminal binary decision diagrams (MTBDDs).

3.3.3 The PRISM Daemon

For simple models PRISM runtime is dominated by Java Virtual Machine initialization. For running multiple model checking tasks in compositional verification, PRISM must be restarted multiple times. Moreover, invoking multiple classic PRISM processes on a grid of computing nodes would require marshaling arguments and remote copying of input files and output results (i.e. ssh and scp). Given the need to scale PRISMATIC to realistically complex systems we enhanced PRISM to act as a web service via a *PRISM daemon*.

The basic design of the PRISMATIC web service is the addition of an embedded web server to PRISM. The PRISMATIC web service can be configured to run on any port using the standard HTTP protocol. In this way multiple instances of the PRISMATIC web service can respond to different requests when configured to listen on different ports. As the PRISMATIC web service acts as a web server it is possible to open its URL in a browser (e.g. `http://localhost:8080`). In this case PRISMATIC displays a form where users can enter command line arguments from a list of predefined switches, free format strings or upload input files. When the user has submitted the form the PRISMATIC output is returned to the browser.

However a more common way to invoke the PRISMATIC web service is from a command line client which marshals arguments and submits input files using multipart encoded forms (using the RFC2388 standard, https://www.ietf.org/rfc/rfc2388.txt). The PRISM output is returned as the standard output of the client and so invoking PRISM in this way is nearly a drop-in replacement for classic PRISM, with the exception that redirecting output to multiple, different files is not permitted.

The benefit of this approach is that the input and output file copying is eliminated, along with the corresponding connection setup and tear down times, as well as the JVM startup time. Even working at a single workstation using the PRISMATIC web service provides a considerable speed increase. In the case of submitting multiple requests to farm of PRISMATIC agents the web service becomes invaluable because it allows the designer to scale each web service server

vertically (on large CPU and RAM machines) or horizontally, and separately the flexibility to host one or multiple web service instances per node.

Our PRISMATIC toolset (notably ABV and PABV) is clever enough to recognize if a PRISMATIC web service URL has been defined and, if so, submit verification tasks directly to the web service.

3.4 Incremental Verification

Incremental verification is an important practical capability for real-world use of the PRISMATIC verification tool. Real-world designs start out imperfect and go through many iterations of a design-verify-revise process. The naive approach is to re-verify the whole system design whenever a bug is fixed or any design decisions have changed. This process is only feasible for small systems, and entirely impractical for the large system-scale verification problems targeted by META.

We have studied techniques for incremental verification that can verify a system property, incorporate a change to part of the system, and then only re-verify as much as needed to ensure that the change of the system does not violate the property. One basis for our work has been the incremental verification techniques that our team developed and patented [24] for nonprobabilistic model checking [25]. By preserving verification information during system design revisions, those methods achieved performance improvements exceeding two orders of magnitude when incorporated into the CIRCA system's automated design-verify-revise process for designing real-time reactive controllers.

In other previous work [7, 37], our team has considered the "component substitutability problem" in the context of evolving software systems. That work focused on being able to replace a modular software system's components and then only locally verify the new component's behavior, using assume-guarantee reasoning to restore confidence in the whole new system. As part of this project, we studied incremental verification and components for the problem domain of cyber-physical systems instead of pure software systems. The new challenges we faced were related to the physical and continuous aspects of cyber-physical systems. In preparation for this, we had already developed compositional verification techniques for hybrid systems [27, 28] and corresponding automatic fixed-point procedures [30, 31]. Recently, we had also developed a verification approach for reconfigurable distributed hybrid systems [29] to help us understand the basis for evolving cyber-physical systems.

Based on these prior developments, we have investigated techniques to decrease the computational cost of iterations in the design and verification flow. The Oxford and SIFT teams implemented incremental verification within PRISMATIC, which reduces the time for re-verification of models when only small, parametric changes to the model are made.

When a design fails initial verification, we expect the designer to make changes and try again. In addition, a designer may want to systematically assess the effects of changes in design parameters. For example, how does overall system reliability change given a range of failure probabilities for one component? However, the computation required can be expensive because model generation and verification have to be performed multiple times. Incremental verification offers improvements in efficiency as it reuses (partial) results from previous verification runs by decomposing the model into strongly connected component groups and retaining calculations for unaffected component groups. Furthermore, the same technique can be used to perform sensitivity

analysis, varying some of the numerical inputs over a small range of values, allowing the developer to reason about estimated parameters such as probabilities or component failure rates.

We extended the PRISM implementation in a number of ways to allow partial recomputation of the models. As mentioned above the first step decomposes the model into strongly connected components. The worst case for tree decomposition is that there will be no decomposition, and hence no savings. Early experience with tree decomposition techniques using randomly generated models produced poor results, but real-world models tend to decompose very well: a sensible model is not randomly generated, but is designed. Experience has shown that real systems decompose extraordinarily well [33, 9].

Given a good tree decomposition and a set of changed variables it is necessary to recompute the model so as to reflect the changes. We modified the treatment of MDP models to support incremental recomputation of the models. Originally, compiled PRISM MDP models contained numbers without any record of what variables were involved in their computation. We extended the MDP model to retain the symbolic expressions representing the transition probabilities so that when a variable is changed the probabilities that need to be recomputed can be recomputed from their expressions. The combination of model decomposition and expression directed model recomputation resulted in significant speedups. While the speedups depend upon the success of the model decomposition algorithm there is good reason to believe that real models (not synthetic) will yield good decompositions that result in dramatic speedups in model recomputation following variable changes.

In summary, for a design tool to be useful, it must be possible for the design engineers to make iterative changes to the model until a model is arrived at that meets design requirements. After each change the models must be updated. For small models this is not a problem but for large models the cost of recompiling the models can be prohibitive yielding a design tool that is too slow to support model debugging by the design engineers. Incremental verification allows small changes to complex models to result in recomputation cost that is proportional to the size of the change, rather than proportional to the size of the model. Our update to the PRISM MDP models has shown that this principle works well when a good model decomposition is obtainable; furthermore there is evidence to support the belief that good decompositions will usually be available for real world models.

3.5 Counterexample Generation and Culprit Identification

When verification fails because a particular system design does *not* satisfy the specified requirements, a verification tool should identify one or more potential *culprits,* or components that lead to the requirements violations [13]. Exact model checking systems search for system execution traces that violate the requirements, and return such a trace as a "counterexample." Any of the design components involved in such a counterexample trace may be the source of the requirements violation.

However, as noted earlier, the standard notion of counterexample does not carry over to probabilistic properties. Unlike in classical model checking, a single execution trace leading to a bad state is often not, by itself, a counterexample to the probabilistic property. Counterexamples to probabilistic properties depend on how likely it is that the system produces requirements-violating execution traces. Thus specific culprit design components may be more difficult to identify.

For example, a system of three parallel components may produce failures in each of those components, each leading to an unacceptable state. If the individual probabilities of those three types of failures are separately below the desired verification threshold, but together lead to an unacceptable probability of failure, then the probabilistic verification will fail. However, no individual failure trace explains the problem or is alone a counterexample to the probabilistic safety requirement. The responsibility for the system-level failure is distributed among the three parallel components.

Recently, several notions of probabilistic counterexamples have been put forward [15, 3]. Essentially, these propose to identify *sets* of failure traces whose combined probability is sufficient to illustrate the violation of a property to be verified. These sets of paths can be generated either using analytic or simulation-based techniques. One of our key PRISMATIC research goals has been to find appropriate ways to capture this type of useful feedback to designers and to devise methods to generate it in an efficient manner. We can then adapt standard machine learning techniques to identify the culprits causing failures. Culprit identification thus consists of four distinct steps:

1. Example and counterexample generation.
2. Summary data extraction.
3. Data analysis.
4. Presentation of culprits.

We have implemented a number of experimental approaches to these steps, which we detail below.

3.5.1 Example and Counterexample Generation

PRISM provides both analytical tools reasoning on the structure of a model, and Monte Carlo-based simulators. We have applied both techniques, in some cases developed specifically for PRISMATIC, to generate details of counterexample traces. PRISM's `-sim` mode was originally oriented to experimental verification of system properties. We extended this mode to export its path information for both example and counterexample generation. To allow experiments to be repeated as unit tests, PRISM allows its random seed to be explicitly specified. PRISM also allows the number of generated paths to be given explicitly or for SPRT mode to automatically determine the minimum number of trials required to prove or disprove a property. Similarly we can specify the maximum length, in the sense of the elapsed time of the simulation, of path to be generated; most of the properties we have tested under our META work have involved time-limited properties, in which case we omit this argument, which becomes optional. As part of the PRISMATIC effort, we have developed the `-simpathv` mode for PRISM simulation to generate path information with low overhead.

For analytic generation of counterexample paths, PRISMATIC uses the techniques put forward in [15]. In order to generate the counterexamples that are most useful for designers, this approach aims to construct a small set of paths by identifying them in descending order of probability, i.e. paths corresponding to the most likely failures are extracted first. Algorithmically, this task is reduced to solving the *k-shortest paths* problem. PRISMATIC implements the REA algorithm of Jiménez and Marzal [20].

3.5.2 Summary Data Extraction

The second step involves preparing a subset or summary of the data in the sample paths. The raw path data itself typically involves too many data to be feasibly examined for culprits, so it is necessary to shrink the data set. We rely here on a property of the models we have considered for META: once the *mode* variable associated with a component transitions from its normal or *nominal* value to some abnormal *off-nominal* value, we know that that mode variable will never change value again. In essence, the abnormal value represents a fatal state for that component. Relying on this assumption allows us to consider only the *final* value of a variable, and so have only one value per variable per run. We can moreover drop any variables whose value is constant — whether nominal or off-nominal — across all runs. Of course, this extraction process is straightforward to extend to any event that is a summary of a single state variable. Expressions describing events that depend on multiple state variables can be specified as an input to `-simpathv` (or added to the model) to include them in PRISM's path output as a single state variable.

We have two implementations of extracting summary data from a set of paths. Both of these implementations use the output of a PRISM run, both generate data in the attribute-relation file format (ARFF).

- The first technique summarizes the value of key path elements as a *multivalued nominal* variable. Each feature in the path summary contains the final value of one component failure indicator. No indication of the semantic interpretation of the value is included in the data (aside from meaningful names for human viewing). That is, we happened to know that, in our test data, certain values indicated failures (abnormal states) of components while other values indicated normal status of those same components, but this information about the meanings of the indicator values is not used in the identification of culprits, and no such knowledge is necessary to use the culprit identification process.
- The second technique summarizes the value of key path elements as a number of *asymmetric binary* variables. A separate variable in the summary corresponds to *each* abnormal value of the key element in the path runs. The underlying assumptions for this technique, and the default key elements, are the same as in the multivalued nominal implementation.

3.5.3 Analysis and Presentation

Currently we have one implemented data analysis, using Waffles to construct a modified decision tree based on the ARFF files. In this analysis, it is assumed that paths that violate the path property indicate the outcomes (e.g., system failures) for which we wish to attribute blame (or credit). The decision tree is built using splits based on information gain, but terminates the splitting process early if the paths in a branch have a sufficient proportion of failures (i.e., paths that violate the path property). Pruning is performed on the tree during construction to eliminate splits if all of the leaf nodes on the resulting branches are assigned the same label. Both the early termination logic and the pruning are important for producing trees from which meaningful culprits can be extracted: without them, the structure of the decision tree tends to obscure the true culprits. There are two parameters associated with Waffles decision tree construction:

- The threshold number of samples below which no further partition of the data will be attempted, corresponding to the `-leafthresh` argument to Waffles.

- The maximum depth of the decision tree, corresponding to the -maxlevels argument to Waffles. By default we use 0, indicating no maximum.

This implementation works with either of the ARFF generators, although our limited testing to this point suggests that the multivalued-nominal implementation produces superior results. This implementation works with either of the ARFF generators, although our limited testing to this point suggests that the multivalued-nominal implementation produces superior results.

The resulting decision tree is not used for classification; instead the information in the structure of the tree is interpreted to ascribe blame to individual components of the system for sets of the instances of failures in the training data. Blame for the failures in each leaf node is ascribed to the last feature used to split the data above that leaf node. The resulting accounting of blame is then presented as a ranked list of suspected features (component failures, in our tests) with the number and relative proportion of failures ascribed to each one. In general, the top culprit is the state variable that is most culpable.

A limitation of this approach is that it assumes that blame can beattributed to individual features. In the event that the property being verified is violated only when multiple events occur (such as the failure of redundant components), the blame for the failure will be assigned to just one of the relevant features. For failure involving symmetric components (e.g., redundant power supplies), the truth (that each of the symmetric components has the same culpability) may be obvious, but in other scenarios it may not be as clear that the top culprit is not solely responsible. However, even in this case the top culprit is likely to be an essential part of the cause. For example, in the case where system failures are caused by the failure of a set of redundant components and just one of them shows up as a culprit, making the identified culprit more reliable will reduce the rate of the system failures as expected, even though there are other ways of addressing the same failures.

We also use Waffles to render the full decision tree as whitespace-formatted monospaced text. This format contains the full information of the tree without interpretation; this may be useful in understanding the list of culprits in some circumstances, but the tree can be very complex and difficult to interpret for systems of even moderate size. We recommend using the summary list of culprits to identify why a particular system property is not satisfied, and referring to the raw tree only when much more detailed information is required. An example of when the full tree might be useful is when it is suspected that a particular culprit is not solely responsible for the system failures ascribed to it. For example, in the case of a system failure caused by failure of a pair of redundant power supplies, inspection of the entire tree would reveal that both of the power supplies had to fail, and not just the single source identified as the culprit.

4.0 RESULTS AND DISCUSSION

4.1 Overview of PRISMATIC Usage

An essential preliminary for incorporating PRISMATIC into the cyber-physical system design process is to compose a functional model of the candidate system. This model is composed of the functional models of the system's components, connected following the topology prescribed in the candidate design. The connecting topology is extracted automatically from the concept description graph, represented in a GXML file, by the XMC tool. XMC generates a PRISM model file, and also generates a file of the target properties of the system in PRISM's PCTL format.

Next, PRISMATIC performs probabilistic model checking of the generated PRISM models. PRISMATIC supports a wide range of probabilistic models including discrete-time and continuous-time Markov chains, Markov decision processes, probabilistic automata, probabilistic timed automata, plus extensions of these models with costs and rewards. Models are described using a simple, state-based language. PRISMATIC provides support for automated analysis of a wide range of quantitative properties of these models, including:

- Internal model consistency (e.g., only one failure mode is active for any component at any time);
- Fault probability analysis (e.g., what is the probability that some fault occurs up to time *T,* or steady state);
- Time bounded functional assessment (e.g., what is the probability that some component is nominal at least up to time *T*);
- Limited-fault analysis (e.g., if exactly *X* component(s) fail(s) by time *T* (or steady state, long run), what is the effect on other failure modes or functions?).

PRISMATIC additionally provides culprit identification: determining, if one of the above properties has insufficiently large probability, the components of the design which are most likely to be responsible. To find culprits, PRISMATIC samples the paths through the model state space to obtain traces associated with both faulting and non-faulting simulations. It collects the values of the mode variables which are off-nominal in any sampled run, along with the boolean evaluation of the property for each sample, and passes them to a decision tree algorithm to identify components whose failures are most often associated with the specified system fault. PRISMATIC reads the culprits from nodes of the decision tree which indicate the target property to be unsatisfied.

The complexity of the analytic algorithms for verifying a system's probabilistic properties is linked to the size of the search space formed by the possible nominal and off-nominal values for each internal mode variable of the components' functional models. Specifically, the number of nodes in the search space is the product of the sizes of these mode variables' domains. PRISM can normally handle search spaces of up to 107–108 nodes on a typical PC, and in certain circumstances up to 1011. The basic underlying verification analyses are essentially polynomial: quadratic or cubic. However, in practice, PRISM uses numerical approximation algorithms to improve this performance. PRISM has been deployed successfully for large, complex applications such as verifying protocols for simultaneous resource access by multiple concurrent agents [23] and rigorous analysis of wireless device discovery protocols [10]. Direct application of these analytic algorithms is unlikely to scale to models of the size anticipated for this program. PRISM use on very large problems has previously benefitted from problem decomposition [10], so to

reach the required scalability in PRISMATIC, we structure the models hierarchically, decomposing each level of the system into a tractable number of subsystems. This work is currently ongoing.

Although XMC's translation of a GXML concept description graph to a PRISM model and property list is automatic, the library of functional models of the basic components must be hand-coded.

The algorithm that builds the decision tree for culprit identification has a worst-case run time of $O(nk^2)$, where n is the number of traces through the model state space and k is the number of components for which a failure mode is defined in the system. The worst case occurs when all component failures happen with equal or nearly-equal probability during both the traces associated with the system fault and the traces from non-fault simulations. In addition, a limit on the depth of the tree can be specified for models with very large numbers of components, which then reduces the scaling to $O(nk)$. In practice, a test case in which 2000 traces are generated from a model with 24 components, all of which experience some failures during the simulations, runs in under 0.2 seconds on a consumer-grade laptop.

Culprit identification requires that the model being examined include states associated with component failures. The current implementation assumes a finite number of nominal failure states for each component. In addition, the process requires a means of identifying the model states associated with component failures; currently it is implemented to depend on the PARC conventions for naming component mode variables.

4.2 OSATE Tool Integration

The Open Source AADL Tool Environment (OSATE) is an Eclipse-based tool for generating, maintaining, and performing analysis on models defined in Architecture Analysis and Design Language (AADL), an SAE standard. We have created a PRISMATIC plug-in for OSATE that:

- Transforms limited forms of AADL system models into PRISMATIC inputs,
- Supports the user in specifying system reliability and probabilistic verification queries,
- Invokes PRISMATIC, and
- Returns the query results to OSATE.

All PRISMATIC features are integrated within the OSATE GUI in the form of menubar buttons. A graphical indicator of the outcome of the PRISMATIC call is annotated to the analyzed file (model) shown in the OSATE navigator pane.

The methodology to transform generic AADL models into a behavioral state machine (BSM) that PRISMATIC can analyze proved to be a significant challenge; ADDL is a highly expressive modeling language that provides the developer with many alternative modes for representing system function and behavior. It was recognized early on that ADDL-to-PRISMATIC translation would need to evolve incrementally, focusing on a subset of model features likely to be associated with systems of interest to META-II. As such, discrete transformation 'strategies' were developed for different AADL model types:

1. AADL models in which the BSM is represented via explicit transitions in the "modes" section of the model's system implementations.

2. AADL models in which the BSM is represented via explicit transitions in the modes section of the model's system implementations and these implementations may have multiple instantiations as "subcomponents."
3. *Default*: AADL models that are augmented with an Error Annex Model (EMA).[1] The EMA supports expression of dependability-related information such as error propagation and stochastic parameters of a system and facilitates describing the BSM relevant to PRISMATIC analysis.

The first two strategies demonstrated simple embedded control system models for which the reliability of a set of input sensors was the key focus. Some key downsides to these strategies that emerged included: 1) The standard AADL does not readily support specification of state transition rates (e.g. from 'operating' to 'failed') so they must be added as meta-comments to the model, and 2) The BSM modeling needed for PRISMATIC analysis may require the developer to modify or insert content in disparate sections of the system AADL model, leading to a scattered representation that can be hard to discern in the overall model.

The effort to scale up to more complex and realistic systems motivated development of the EMA-based strategy, and it proved to be the most versatile and least intrusive to integrate with the system's functional modeling. It resolves both of the above-mentioned shortcomings by allowing development of a discrete BSM model in the EMA library associated with an independently developed AADL system model. Successful demonstrations of this transformation strategy prepared for the July PI meeting involved PRISMATIC analysis of two different portions of a realistic engineered system, Rockwell Collins' Air Data System.

One difficulty with the EMA-based strategy was the lack of support for the Error Model Annex in OSATE v2, which was released in June 2010 to handle the AADL 2.0 standard. In order to leverage the benefits of the EMA we developed a tool for converting AADL-2 models (such as the Rockwell-Collins ADS) to AADL-1 and then used OSATE v1.5.8 augmented with the PRISMATIC plugins. This shortcoming should soon be resolved as SEI, the developer of AADL and OSATE, is incrementally releasing versions of both AADL-2 and OSATE v2 that fully support the EMA.

4.3 External Verification of OpenModelica Models

Modelica is a language for describing systems, both discrete state machines such as Stateflow, and continuous dynamics such as Simulink. OpenModelica is an evolving toolkit for constructing and simulating these models. CMU delivered a toolkit that applies Bayesian techniques to the problem of statistical modeling checking for systems with continuous dynamics, by gathering sample execution traces. SIFT integrated this toolkit with Modelica simulations and an automatic mechanism to vary model component parameters (e.g., the nominal resistance of a resistor in a circuit). The result is a compact, self-contained toolkit that can be used to assess probabilistic bounded LTL hypotheses on continuous- or hybrid-dynamics models captured in Modelica.

---

[1] Dependability Modeling with the Architecture Analysis & Design Language (AADL) http://www.sei.cmu.edu/library/abstracts/reports/07tn043.cfm

4.3.1 RLC Model

Our initial goal with OpenModelica has been to model the example RLC circuit provided by Dr. Eremenko, beginning with its relatively simple electrical properties and eventually adding thermal effects. The RLC circuit is set up as a low-pass filter (possibly bandpass), meaning that lower input frequencies should pass relatively unchanged through the circuit, while higher frequencies should be filtered out (have the output voltage decreased). Electrical engineering techniques can analytically assess the expected performance of the circuit. With OpenModelica, we must use simulation. We are developing methods to run numerous simulations of the circuit under different experimental conditions, such as varying the input voltage source's sinusoidal frequency. We assess the output to ensure that the simulation is providing the expected behavior. Example properties include:

- **Max (and min) voltages** — Assessing the peak values ensures that the circuit outputs, for example, no more voltage than is input.
- **Steady-state times** — How quickly the circuit settles to steady-state behavior.
- **Transient response** — How the circuit responds to impulse inputs (e.g., rise time to some output voltage threshold, settling time back to a low output voltage, and whether it is under/over/critically damped).

More realistic properties might include the cutoff or corner frequency where the output of the system is down 3 dB, its performance under variations in the component parameters, and its phase shift. The problem with the latter property for simulation is that it is a property not just of many simulations, but is an analysis/property of the simulations that is not directly supported by a pure temporal logic such as LTL. Resolving this difficulty would require a way to run a tool such as a curve fitter on the data, extract the frequency/phase/amplitude information, and assess its results against numeric values specified in the property. Another approach to analyzing this circuit in Modelica would be to simply input the equations that describe the analytic expected behavior, but this approach may not enable significantly different analyses.

Figure 4 shows our first attempt at modeling the RLC circuit using the Modelica language and the OpenModelica OMEdit graphical model editing tool, with plots of circuit simulation results at various frequencies in Figures 5 through 7.

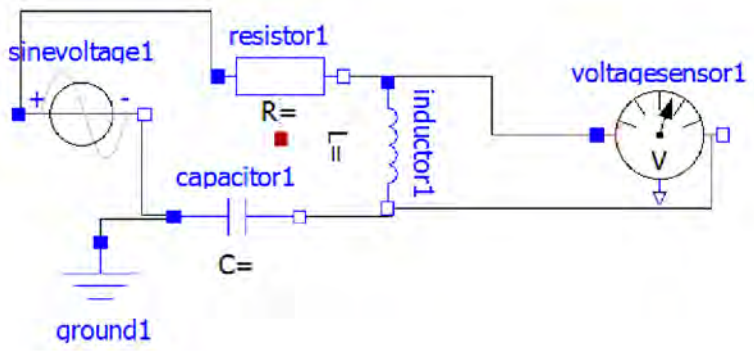

Figure 4: Example RLC Circuit, Modeled in OpenModelica's OMEdit

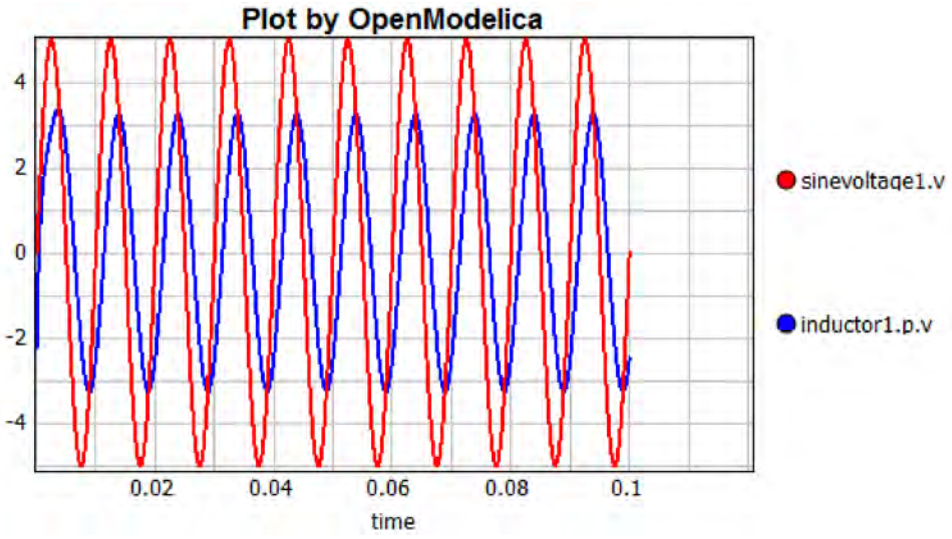

Figure 5: Example OMEdit Plot of RLC Circuit Simulation Results with 100-Hz Input

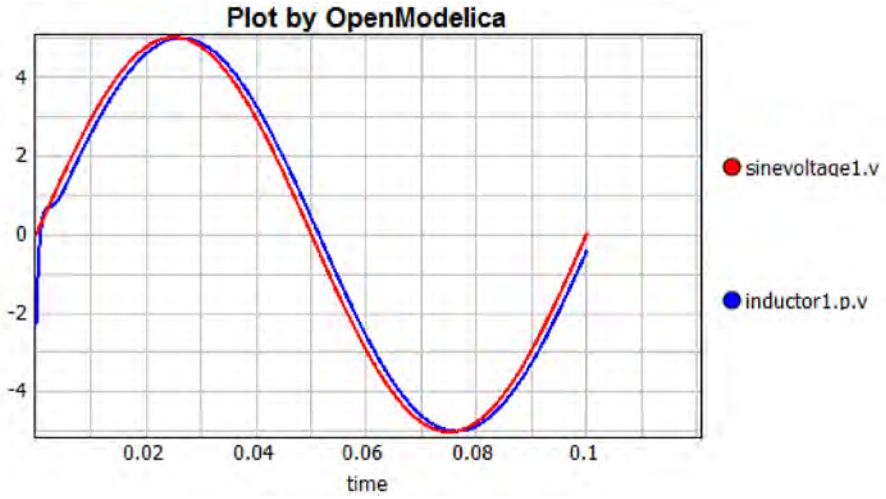

Figure 6: Example OMEdit Plot of RLC Circuit Simulation Results with 10-Hz Input

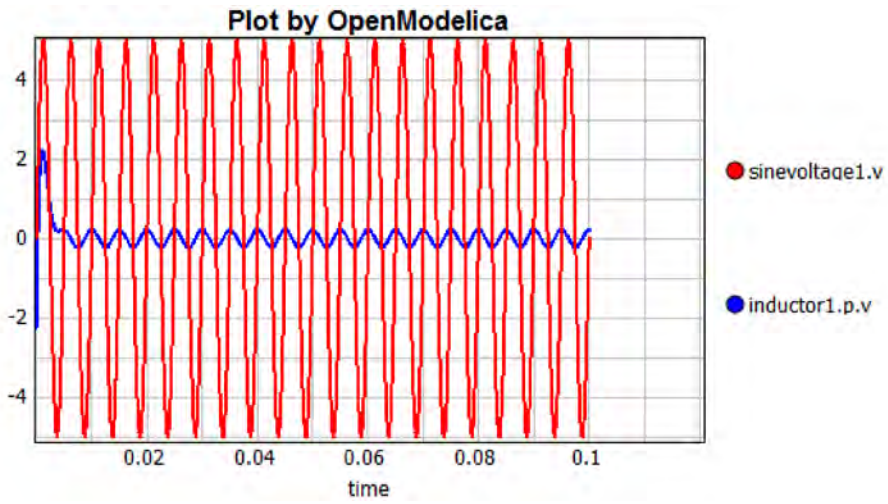

Figure 7: Example OMEdit plot of RLC Circuit Simulation Results with 200-Hz Input

4.4 Function Failure Logic Modeling

Function failure logic models are used to capture the nominal and failure modes of components and how they interact through exchanges of abstract or qualitative "flows" corresponding to energy (e.g., electrical, mechanical, thermal), information (e.g., sensor signals), and other influences.

For example, at one level of abstraction a vehicle powertrain could be modeled as a relatively simple combination of engine/motor, transmission, driveshaft, differential, and wheels or tracks. Each of these components would have a nominal operating mode, with other state variables representing different sub-states such as the selected gear of the transmission and the motor's throttle level (discretized). Flow expressions would represent the movement of fuel into the engine, and mechanical energy (torque/power) from the engine through to the wheels or tracks. Models of the environment may also be included, to capture interactions between the final drive and the ground surface (e.g., various levels of traction/slipping). Thus the nominal component behaviors will support reasoning at a coarse level about how the powertrain operates normally, and can be used to rapidly verify various normal-operation requirements such as "The powertrain shall support speeds up to 40 miles per hour" and "The powertrain shall support missions at least 10 hours long at half-throttle without refueling."

Components typically also have off-nominal modes in which their behaviors are degraded or failed entirely. These off-nominal modes can affect the flows between components so that the model can be used to reason about the performance of the overall powertrain when, for example, the transmission can no longer achieve third gear. These failure modes have associated probabilities (likelihoods or failure rates), and thus support probabilistic verification of system-level properties. So, rather than simply verifying that a powertrain can support a 10-hour mission without refueling, PRISMATIC could prove how likely the system is to satisfy that requirement, accounting for the various failure modes, their probabilities, and their impact on overall system behaviors.

4.5 Scalability Assessment

We have evaluated the performance of PRISMATIC on both real-world cyber-physical system designs and synthetic, scalable cyber-physical system designs. Real-world component models and system designs may be drawn from both our prior work on verification and cyber-physical systems, as well as other META participants. The government may choose to provide cyber-physical system designs or link our team with other performers who are working in related areas such as meta-modeling or design tools. Our project plan includes attending joint PI meetings and other coordination meetings as necessary to cooperate actively with other contractors.

We can draw on several previous projects to supply existing system designs. For example, the PRISM case study repository [32] contains approximately 50 large case studies that will be used to evaluate the underlying techniques developed for PRISMATIC. These case studies cover a wide range of application domains including, for example, wireless communication systems, communication protocols and power management schemes.

To conduct more controllable evaluations of PRISMATIC's performance and scalability, we have experimented with synthetically-generated cyber-physical system designs that can be scaled to various sizes. We used the Relay component from our model library to synthesize an arbitrarily complex system design of Relays in series. We then leveraged the new web service feature of PRISMATIC by depoying it on a grid of could computing nodes inside Amazon Web Service's Elastic Computing Cloud (EC2) infrastructure.

The approach to creating the complex system design involves leverages the compositional verification technique (described above in Section 3.3) to create an "abstract" system which will represent a set of N components. We then repeat this process for M levels of system hierarchy to quickly create very large models. For example when each decomposition involves M=5 levels of hierarchy with N=7, components each have a system of 19,608 components: 2,801 are abstractions and 16,807 are detailed leaf nodes.

In order to verify the entire system we applied our abstraction verification tool (ABV) to each of these 2801 abstractions *in parallel* on 64 EC2 servers. This process took 1.9 hours and a cost of $2.58. After some architectural refinements we submitted an even larger synthetic model for verification: again 5 levels of hierarchy, but this time with 9 components each. This design comprises 66,430 components: 7,381 of which are abstractions and 59,049 are detailed leaf nodes. Deployed in 16 larger EC2 servers this verification took 1.25 hours and a cost of $40.52. Thus the composition of our PRISMATIC web service with the compositional verification technique demonstrates that PRISMATIC can scale to model realistically large systems.

5.0 CONCLUSIONS

In the Phase 1 effort, we have developed the PRISMATIC tool to efficiently perform several key probabilistic verification functions on complex cyber-physical system designs. PRISMATIC verifies the link between system designs and their formalized requirements and can actively guide designers to appropriate system modifications when requirements are not met. In particular:

- Our work on compositional and incremental verification on the scalability of verification for PRISMATIC expands the reach of formal verification technologies, allowing PRISMATIC to verify extremely complex systems and scale well with model size.
- PRISMATIC verifies new system designs quickly and takes advantage of parallel processing, exploiting the parallelizability of statistical methods and supporting concurrent verification of numerous system properties.
- PRISMATIC's compositional and incremental verification techniques support rapid re-verification after changes to a cyber-physical system design, minimizing the need to re-verify components that are not changed.
- PRISMATIC provides not only verification of system properties, but moreover helps guide debugging and system redesign efforts by identifying culprits and by deriving requirements on future design revisions that will move a system closer to compliance with desired safety or behavioral specifications.
- PRISMATIC's statistical verification methods work for any type of system that can be simulated, using any form of probability distributions. Thus, PRISMATIC easily combines results from different cyber-physical design disciplines.

The impact of PRISMATIC on cyber-physical system design will be to make formal verification an everyday part of the design process: it is a practical tool that designers can use frequently as part of the daily iterative design cycle. PRISMATIC can dramatically reduce the need to physically build and test system components to ensure proper operations. When such real-world tests are conducted, PRISMATIC's statistical verification methods help minimize the testing required.

6.0 REFERENCES

1 MATLAB® Simulink®/Stateflow. The MathWorks™, Inc., http://www.mathworks.com (accessed Sept 27, 2011).

2 NESSUS Overview. Southwest Research Institute. http://www.nessus.com, (accessed Sept 27, 2011).

3 Aljazzar, H. and Leue, S. "Directed Explicit State-space Search in the Generation of Counterexamples for Stochastic Model Checking," *IEEE Transactions on Software Engineering,* 36(1):37–60, January 2010.

4 Alur, R., Courcoubetis, C. and Dill, D., "Model-checking for Probabilistic Real-time Systems, in Proc. 18th ICALP*,* pp. 115–126, 1991.

5 Baier, C., Haverkort, B.R., Hermanns, H., and Katoen, J.-P., "Model-checking algorithms for continuous-time Markov Chains," *IEEE Transactions on Software Engineering,* 29(6):524–541, 2003.

6 Chaki, S., Clarke, E., Giannakopoulou, D., and Pasareanu, C.S., Abstraction and Assume-guarantee Reasoning for Automated Software Verification, Technical Report 05.02, NASA, 2004.

7 Chaki, S., Clarke, E.M., Sharygina, N., and Sinha, N., "Verification of Evolving Software via Component Substitutability Analysis," *Formal Methods in System Design,* 32(3):235–266, 2008.

8 Clarke, E.M. and Zuliani, P., "Statistical Model Checking for Cyber-physical Systems, in ATVA 2011: 9th International Symposium on Automated Technology for Verification and Analysis, Taipei, Taiwan, Oct. 2011. (To appear)

9 De Givry, S. and Schiex, T., "Exploiting Tree Decomposition and Soft Local Consistency in Weighted CSP," in Proceedings of AAAI. 2006.

10 Duflot, M., Kwiatkowska, M., Norman, G., and Parker, D., "A Formal Analysis of Bluetooth Device Discovery," *International Journal on Software Tools for Technology Transfer (STTT),* 8(6):621–632, Nov. 2006.

11 Feng, L., Kwiatkowska, M., and Parker, D., "Compositional Verification of Probabilistic Systems using Learning," in Proc. 7th International Conference on Quantitative Evaluation of Systems (QEST'10). IEEE CS Press, 2010. To appear.

12 Gheorghiu Bobaru, M., Pasareanu, C.S., Giannakopoulou, D. "Automated Assume-Guarantee Reasoning by Abstraction Refinement," in Gupta, A., Malik, S. (Eds.): Computer Aided Verification, 20th International Conference, CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proceedings. Lecture Notes in Computer Science 5123, pp. 135–148, Springer, 2008.

13 Goldman, R.P., Pelican, M.J.S., and Musliner, D.J., "Guiding Planner Backjumping using Verifier Traces," in Proc. International Conference on Automated Planning and Scheduling*,* 2004.

14 Grosu, R. and Smolka, S., 'Monte Carlo Model Checking," in TACAS*,* Vol. 3440 of Lecture Notes in Computer Science*,* pp. 271–286, 2005.

15 Han, T., Katoen, J.-P., and Damman, B., "Counterexample Generation in Probabilistic Model Checking," *IEEE Transactions on Software Engineering,* 35(2):241–257, 2009.

16 Hansson, H. and Jonsson, B., "A Logic for Reasoning about Time and Reliability," *Formal Asp. Comput.,* 6(5):512–535, 1994.

17 Hérault, T., Lassaigne, R., Magniette F., and Peyronnet, S., "Approximate Probabilistic Model Checking,". in VMCAI*,* Vol. 2937 of Lecture Notes in Computer Science, pp. 73–84, 2004.

18 Hermanns, H., Hahn, E.M., Wachter, B., and Zhang, L., "Time-bounded Model Checking of Infinite-state Continuous-time Markov Chains," *Fundamenta Informaticae,* 95(1):129–155, 2009.

19 Hinton, A., Kwiatkowska, M., Norman, G., and Parker, D., "PRISM: A Tool for Automatic Verification of Probabilistic Systems," in TACAS*,* Vol. 3920 of Lecture Notes in Computer Science, pp. 441–444, 2006.

20 Jiménez, V.M., Marzal, A., "Computing the k shortest paths: a new algorithm and experimental comparison," Proc. Workshop Algorithmic Engineering, p 15-29, 1999.

21 Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., and Zuliani, P., "A Bayesian Approach to Model Checking Biological Systems," in Proc. Computational Methods in Systems Biology (CMSB'09), 2009.

22 Kwiatkowska, M., Norman, G., Parker, D., and Qu, H., "Assume-guarantee verification for probabilistic systems," in Proc. 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'10), Lecture Notes in Computer Science, Springer, 2010.

23 Kwiatkowska, M., Norman, G., and Segala, R., "Automated Verification of a Randomised Distributed Consensus Protocol using Cadence SMV and PRISM," in Berry, G., Common, H., and Finkel, A.(Eds.), Proc. CAV'01, No. 2102 in Lecture Notes in Computer Science, pp. 194–206, January 2001.

24 Musliner, D.J., Goldman,R.P., and Pelican, M.J., "Incremental Automata Verification," United States Patent 6957178, October 2005.

25 Musliner, D.J., Pelican, M.J.S., and Goldman, R.P., "Incremental Verification for On-the-fly Controller Synthesis," *Electronic Notes in Theoretical Computer Science*, 149(2), Feb. 2006.

26 Pasareanu, C., Giannakopoulou, D., Bobaru, M., Cobleigh, J., and Barringer, H., Learning to Divide and Conquer: Applying the L* Algorithm to Automate Assume-guarantee Reasoning, *Formal Methods in System Design,* 32(3):175–205, 2008.

27 Platzer, A., "Differential dynamic logic for hybrid systems," *Journal of Automated Reasoning,* 41(2):143–189, 2008.

28 Platzer, A., "Differential-algebraic Dynamic Logic for Differential-algebraic Programs," *J. Log. Comput.,* 20(1):309–352, 2010. Advance Access published on November 18, 2008.

29 Platzer, A., "Quantified Differential Dynamic Logic for Distributed Hybrid Systems," in Dawar, A. and Veith, H., (Eds.), CSL, Lecture Notes in Computer Science. Springer, 2010.

30 Platzer, A. and Clarke, E.M., “Computing Differential Invariants of Hybrid Systems as Fixed Points,” in Gupta A. and Malik, S. , (Eds.), CAV, Vol. 5123 of Lecture Notes in Computer Science, pp. 176–189, Springer, 2008.

31 Platzer, A. and Clarke, E.M., “Computing Differential Invariants of Hybrid Systems as Fixed Points,” *Form. Methods Syst. Des.,* 35(1):98–120, 2009.

32 PRISM web site: http://www.prismmodelchecker.org/, (accessed Sept 27, 2011).

33 Sachenbacher, M. and Williams, B.C., “Solving Soft Constraints by Separating Optimization and Satisfiability,” Proceedings of SOFT’05, 2005.

34 Schmitt, E., “Crash Destroys F-22 Test Model,” *The New York Times,* April 29, 1992.

35 Segala, R. Lynch, N., “Probabilistic Simulations for Probabilistic Processes,” *Nordic Journal of Computing*, 2(2):250-273, 1995. An extended abstract appears in Proceedings of the 5th International Conference on Concurrency Theory (CONCUR '94), Uppsala, Sweden, LNCS 836, pages 481-496, August 1, 1994.

36 Sen, K., Viswanathan, M., and Agha, G., “Statistical Model Checking of Black-box Probabilistic Systems,” in CAV, Vol. 3114 of Lecture Notes in Computer Science, pp. 202–215, 2004.

37 Sharygina, N., Chaki, S., Clarke, E.M., and Sinha, N., “Dynamic Component Substitutability Analysis, in Fitzgerald, J., Hayes, I.J., and Tarlecki, A. (Eds.), FM, Vol. 3582 of Lecture Notes in Computer Science, pp. 512–528, Springer, 2005.

38 Slabodkin, G., “Navy: Calibration Flaw Crashed Yorktown LAN,” *Government Computer News,* Nov. 1998.

39 Younes, H.L., Kwiatkowska, M., Norman, G., and Parker, D., “Numerical vs. Statistical Probabilistic Model Checking,” *International Journal on Software Tools for Technology Transfer,* 8(3):216–228, 2006.

40 Younes, H.L. and Musliner, D.J., “Probabilistic Plan Verification through Acceptance Sampling,” in Proc. AIPS-02 Workshop on Planning via Model Checking, pp. 81–88, Apr. 2002.

41 Younes, H.L.S. and Simmons, R.G., “Probabilistic verification of discrete event systems using acceptance sampling,” in Proc. 14th International Conference on Computer Aided Verification, Vol. 2404 of Lecture Notes in Computer Science, pp. 223–235, Springer, 2002.

42 Younes, H.L.S. and Simmons, R.G., “Statistical Probabilistic Model Checking with a Focus on Time-bounded Properties,” *Information and Computation,* 204(9):1368–1409, 2006.

43 Zuliani, P. and Clarke, E.M. “Statistical Model Checking of Rare Events in Stochastic Hybrid Systems,” submitted to DATE 2012: Design, Automation and Test in Europe.

44 Zuliani, P., Platzer, A., and Clarke, E.M., “Bayesian Statistical Model Checking with Application to Simulink/Stateflow Verification,” in Johansson, K.H. and Yi, W. (Eds.), HSCC, pp. 243–252. ACM, 2010.

.

List of Symbols, Abbreviations, and Acronyms

AADL	Architecture Analysis and Design Language
ABV	Abstraction Verification Tool
AFRL	Air Force Research Laboratory
AGAR	Assume-Guarantee Abstraction Refinement
ARFF	Attribute-Relation File Format
BDD	Binary Decision Diagram
BSM	Behavioral State Machine
BT	British Telecom
CMU	Carnegie Mellon University
CSP	Communicating Sequential Processes
DARPA	Defense Advanced Research Projects Agency
EC2	Amazon Web Service's Elastic Computing Cloud
EMA	Error Model Annex
EPSRC	Engineering & Physical Sciences Research Council (UK)
ERC	European Research Council
ETMCC	Erlangen-Twente Markov Chain Checker, a prototype model checker for continuous-time Markov chains and a predecessor to MRMC
MDP	Markov Decision Process
MRMC	Markov Reward Model Checker. A model checker for discrete-time and continuous-time Markov reward models that can consume model data exported by PRISM
MTBDD	Multi-Terminal Binary Decision Diagrams
OSATE	Open Source AADL Tool Environment
PABV	Parallel Abstract Verifier
PEPA	Performance Evaluation Process Algebra
PMC	Probabilistic Model Checking
SBML	Systems Biology Markup Language
SIFT	Smart Information Flow Technologies—a small research company specializing in intelligent automation and human-centered systems
SMT	Satisfiability Modulo Theories. SMT solvers generalize SAT solvers to arithmetic and other formal theories.
SPRT	Simple Packet Relay Transport
VLSI	Very Large-Scale Integration

Glossary

APMC	A statistical probabilistic model checker that uses the PRISM modeling language
GXML	Markup language we use for concept description graphs
INFAMY	A research prototype model checker for probabilistic systems that uses the PRISM modeling language
LTL	A pure temporal logic
MATLAB® Simulink®	A simulation tool.
META	Program that aims to improve the process of building cyber-physical systems by developing new model-based design flows and tools that can capture all functional and logical aspects of a system design, allowing design-time verification of system behavioral properties
Modelica	A language for describing systems
NESSUS	A probabilistic analysis tool for structural and machanical systems
OpenModelica	A toolkit for Modelica
PARAM	A research prototype model checker for probabilistic systems that uses the PRISM modeling language
PASS	A research prototype model checker for probabilistic systems that uses the PRISM modeling language
PCTL	PRISM file format for describing target properties
PRISM	*Probabilistic model checker*, a tool for formal modeling and analysis of systems that exhibit random or probabilistic behavior; developed by the University of Oxford
PRISMATIC	Unified tool and technique for formal design verification to address the challenges of verifying complex cyber-physical system designs before manufacturing and testing
SEI	Developer of AADL and OSATE
Simulink	A representation of continuous dynamics
SAT	The Boolean satisfiability problem
StateFlow	A representation of discrete state machines
XMC	Part of PRISM, converts GXML to PRISM and PCTL
Yices	An SMT solver
YMER	A statistical probabilistic model checker that uses the PRISM modeling language